**1-General Introduction**

Bushfires represent an increasingly serious and destructive threat, causing major environmental, economic, and human damage every year. This issue is particularly acute in West Africa, where drought periods intensify both the frequency and severity of these fires. In response, the need for a reliable early detection and rapid alert system has become critical. The **FireGuard** system was developed in this context, offering an innovative solution based on the Internet of Things (IoT) to improve how such disasters are managed. This technical report presents the complete design of the system-both hardware and software its architecture, and its main features. Special attention is given to the use of **ESP32 Wi-Fi communication** for efficient data transmission, enabling autonomous surveillance of at-risk areas and fast alert delivery.

**2. System Specifications**

**2.1. Functional Requirements**

The FireGuard system was designed to meet a specific set of functional requirements aimed at ensuring early detection, effective alerting, and continuous monitoring of bushfires. These requirements define the core operational capabilities of the system:

Environmental Data Acquisition: The system continuously measures ambient temperature, relative humidity, and the presence of gases (carbon monoxide and other combustion/smoke gases) from each monitoring node.

Node Geolocation: Each monitoring node determines and transmits its precise geographic coordinates (latitude and longitude) in real time.

Reliable Data Transmission: Sensor readings and GPS coordinates collected by the nodes are wirelessly transmitted via Wi-Fi to a centralized access point or collector in a reliable and periodic manner.

Data Collection and Centralization: The access point aggregates and relays all data from the nodes to a centralized storage and analysis platform.

Persistent Data Storage: All raw and processed sensor data is stored persistently and in a structured format using Google Sheets, allowing future access and analysis.

Anomaly Detection and Analysis: The system analyzes incoming data in real time to detect fire-triggering conditions (e.g., exceeding critical temperature thresholds, simultaneous detection of smoke/gas) using Google Apps Script.

Automatic Alert Generation: When a critical situation is detected, the system automatically generates and triggers alerts in both the database and the monitoring dashboard.

Intuitive Data Visualization: An interactive dashboard provides real-time visualization of the status of all monitoring nodes, displaying the latest sensor readings and geographic positions on a map.

Data History and Trend Analysis: The dashboard allows access to historical sensor data through charts and graphs for trend analysis.

Visual Alert Indicators: The dashboard clearly highlights active alerts and their precise locations through visual markers.

**2.2. Non-Functional Requirements / Constraints**

The design and implementation of the FireGuard system incorporate several non-functional requirements and constraints that define the prototype’s quality, performance, and viability. These elements are essential to ensure the system’s effectiveness and operability in its intended environment:

Measurement and Transmission Frequency: The system collects and transmits sensor data every 6 seconds, ensuring near real-time detection of environmental changes.

Prototype Budget Constraint: The development and assembly of the prototype were limited to a maximum budget allocated for this **Africa Deep Tech Challenge 2025**, requiring cost-effective component selection.

Energy Autonomy: Each sensor node is energy-autonomous, powered by an integrated solar system, reducing the need for frequent human intervention for recharging or battery replacement.

Robustness and Environmental Resistance: Although the sensor enclosures are 3D-printed in PLA, they provide protection against basic weather conditions (dust, light humidity) and minor shocks, making them suitable for outdoor deployment.

Data Accuracy: The system delivers reliable measurements (temperature, humidity, gas, GPS position) with sufficient precision to support accurate environmental assessments and threat localization.

Communication Reliability: The Wi-Fi connection between the nodes and the central collector ensures stable data transmission and manages automatic reconnection in case of disruptions.

System Latency: The entire data processing chain from sensor capture to dashboard display and alert triggering is designed to maintain minimal latency for rapid response.

Data Security: Data transmission is handled in a way that limits risks of unauthorized interception or modification.

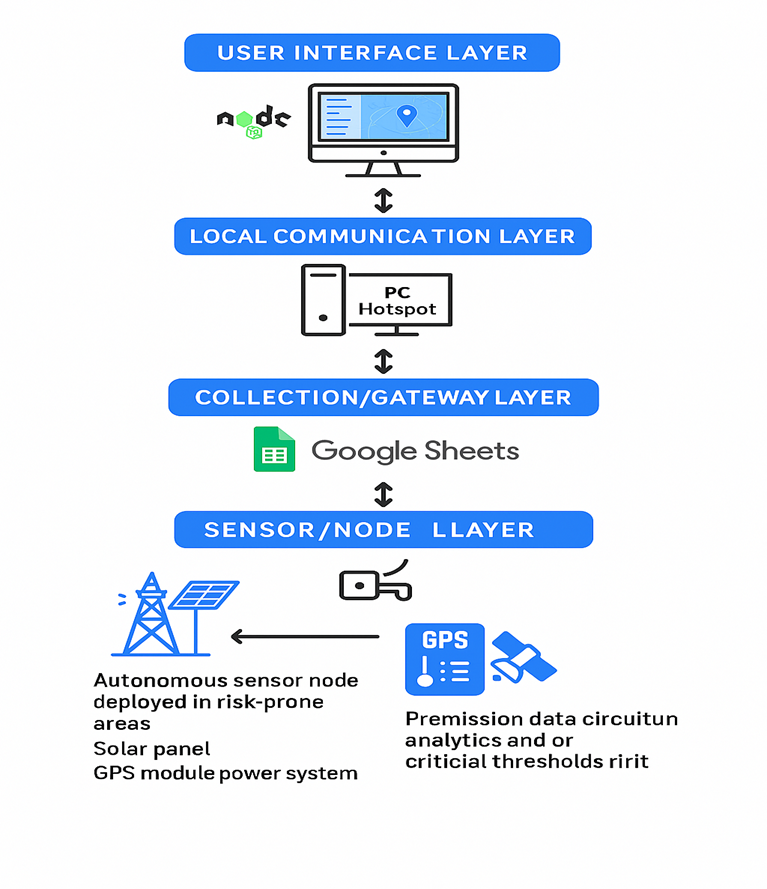
Ease of Deployment and Maintenance: The node design allows for easy field installation and simplified maintenance operations.

Scalability: The system architecture is designed to support the future integration of additional nodes, extra sensors (such as the CO₂ sensor not included in the current prototype), and improvements to the analysis algorithms.

**3. Detailed System Architecture**

**3.1. Implemented General Architecture**

This section presents the overall structure of the FireGuard system, its main functional layers, and the data flow between them.



The system architecture is designed to support a complete value chain, from field detection to centralized visualization and alerting. It is structured around several interconnected layers that ensure the continuous flow of data.

* Sensor/Node Layer: This foundational layer consists of autonomous monitoring nodes deployed in high-risk areas. Each node is equipped with an ESP microcontroller (featuring an integrated SIM800L module, though unused in the current transmission setup), a set of environmental sensors (BME for temperature and humidity, MQ-2 for smoke/gas, MQ-7 for CO), a GPS module for geolocation, and a solar-powered energy system (20W panel, LM7805S, TP4056, Li-ion battery) ensuring energy autonomy. The role of these nodes is to collect raw data every 6 seconds.
* Local Communication Layer: Data transmission from the sensor nodes is handled via the ESP’s built-in Wi-Fi connectivity. Each node connects to the Wi-Fi network created by the central access point/collector to send its measurements.
* Collection/Gateway Layer: In the prototype, this layer is represented by a personal computer (PC) configured as a Wi-Fi hotspot. The PC acts as the central access point, receiving data via HTTP POST requests sent by all monitoring nodes. It serves as a gateway, collecting this information before forwarding it to the cloud storage and analysis layer.
* Cloud Storage/Analysis Layer: The data collected by the PC hotspot is sent to Google Sheets, which functions as the system’s online database. Data reception and integration into the spreadsheets are managed through Google Apps Script, which also performs real-time analysis to detect critical thresholds and trigger alerts accordingly.
* User Interface Layer: This top layer is a web dashboard developed with Node.js. It interacts with Google Sheets (via the Google Sheets API or custom endpoints) to retrieve both real-time and historical sensor data. The dashboard displays this information in a visual and intuitive manner, including the geographical location of nodes on an interactive map, sensor data graphs, and active alerts.

**Overall Data Flow:**

The data flow begins with sensor readings collected by each ESP node. These timestamped and geolocated measurements are transmitted via Wi-Fi to the PC hotspot. The PC software receives the data and pushes it to Google Sheets, where Google Apps Script handles analysis and alert detection. Finally, the Node.js dashboard queries Google Sheets to present system status and alerts to the end user.

**3.2. Hardware Architecture of the Nodes**

This section details the specific hardware components selected to build a monitoring node and how they are interconnected.

|  |  |  |
| --- | --- | --- |
| **Component** | **Reference / Type** | **Functionality / Role in the System** |
| Microcontroller | ESP Board with integrated SIM800L (e.g., Lilligo/Tilligo type) | The node's brain, manages sensor readings, data processing, Wi-Fi connection, and power management. The SIM800L function is present but not used for current transmission; it is a future improvement perspective. |
| Temperature/Humidity/Pressure Sensor | BME (e.g., BME280) | Precisely measures ambient temperature, relative humidity, and atmospheric pressure, providing essential data to assess environmental conditions and predict fire risks. Temperature data is crucial for alert triggering: if temperature > 45°C, the system signals a " Danger: High Temp". |
| Carbon Monoxide Sensor | MQ-7 | Specifically detects the presence of carbon monoxide (CO). Its reading is used for alert detection: if the co > 3500 value is exceeded, the system signals a " Danger: CO". |
| Smoke/Flammable Gas Sensor | MQ-2 | Detects a wide range of flammable gases (LPG, methane, hydrogen) and smoke. Its reading is used for alert detection: if the gas > 4000 value is exceeded, the system signals an " Danger: Gas/Smoke". |
| GPS Module | (e.g., Ublox NEO-6M) | Provides precise geographical coordinates (latitude, longitude) of the node, allowing accurate localization of the alert origin on the dashboard map. |
| Solar Panel | 20 Watts, 18 Volts | Primary renewable energy source, converts light energy into electrical energy to power the node and recharge the battery, ensuring the system's autonomy outdoors. |
| Voltage Regulator | LM7805S | Lowers the 18V voltage from the solar panel to a stable 5V, necessary to power the battery charging module and potentially other components of the node. |
| Battery Charger Module | TP4056 | Manages the charging of the Li-ion battery securely (protection against overcharge and over-discharge) using the regulated energy supplied by the solar panel. |
| Battery | Li-ion (e.g., 3.7V nominal, 3.37V measured) | Stores electrical energy generated by the solar panel, ensuring continuous power to the node, including during periods without sunlight (night, bad weather). |
| Enclosure | PLA (3D printed) | Physical housing for electronic components, protects all internal elements from basic environmental factors and minor shocks. The design is optimized for component integration and sensor ventilation. |

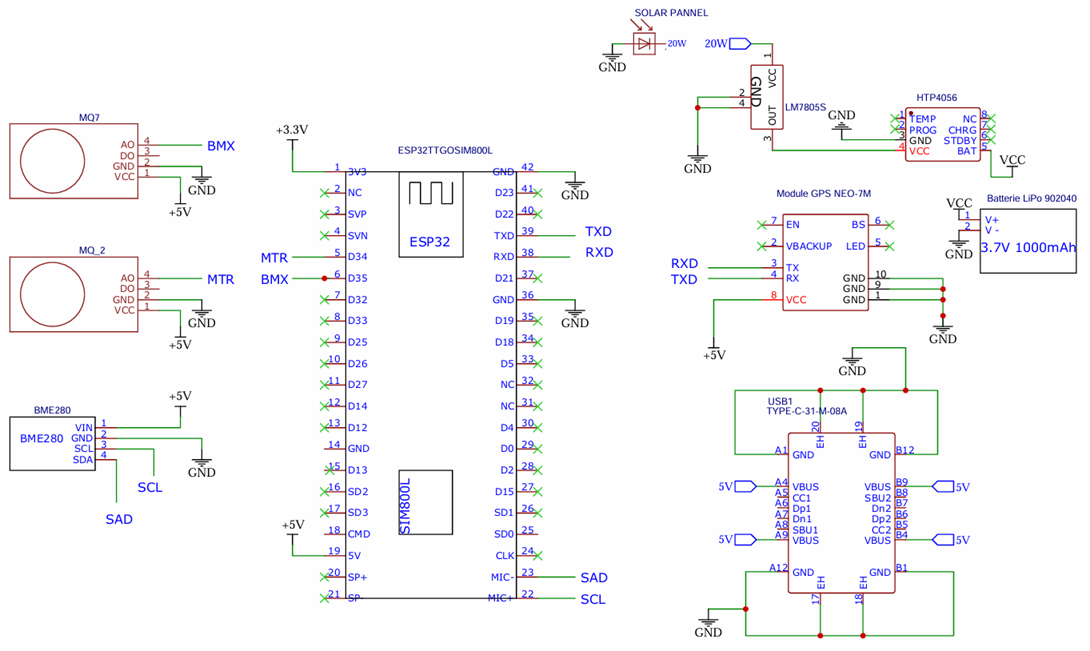
**Pins and Connections:**

In each monitoring node, integrating the sensors with the ESP microcontroller required specific wiring. The BME sensor (for temperature, humidity, and pressure) is connected via the ESP’s I2C interface. The MQ-7 (carbon monoxide) and MQ-2 (smoke/flammable gases) sensors, being analog, are connected to the ESP’s analog input pins for reading their values. The GPS module communicates with the ESP through the UART (serial communication interface).

The power module, which includes the solar panel, the LM7805S voltage regulator, and the TP4056 charging module, is wired to supply the necessary power to the ESP and sensors while also managing the charging of the Li-ion battery.

Protection Circuit:

To protect the battery and node components, the TP4056 module includes essential protection circuits against overcharging and deep discharge of the Li-ion battery. For the purposes of this prototype, developed during the **Africa Deep Tech Challenge 2025**, this integrated protection is considered sufficient.



This diagram illustrates the detailed wiring schematic of the monitoring node, showing the interconnections between the ESP microcontroller, the sensors, the GPS module, and the power supply system.

**4. General Presentation of the Platform**

As part of this early bushfire detection project, my main responsibility focused on the design, development, and deployment of the web platform used to visualize and monitor in real time the data collected by the IoT sensors.

This platform plays a key role in the overall system: it serves as the interface between the raw data collected from the sensors and the end users (monitoring teams, local authorities, technicians). Through this interface, it becomes possible to track environmental conditions, locate high-risk areas, and respond quickly when danger is detected.

**4.1. Monitoring Node Firmware**

The firmware embedded on each ESP microcontroller in the monitoring nodes is the cornerstone for data collection and transmission.

* **Development Environment and Programming Language:**

The firmware is developed using the Arduino IDE environment, programmed in C++. This allows leveraging specific libraries for the ESP and sensors.

* **Libraries Used:**

The code integrates several standard and specialized libraries to handle various functionalities:

WiFi.h: Manages the ESP’s Wi-Fi connectivity.

HTTPClient.h: Handles sending HTTP POST requests to the server.

Wire.h: Manages I2C communication with the BME280 sensor.

Adafruit\_Sensor.h and Adafruit\_BME280.h: Provide interface and data reading for the BME280 sensor.

* **Operation Logic:**

Initialization: During startup (setup()), the microcontroller initializes serial communication, connects to the Wi-Fi network defined by SSID and password (e.g., "UNIPOD-STUDIO-1 2304"), and initializes the BME280 sensor (I2C address 0x76). Pins for MQ-7 (pin 35) and MQ-2 (pin 34) sensors are configured as INPUT.

Sensor Reading**:** In the main loop (loop()), the firmware reads sensor data: temperature and humidity using bme.readTemperature() and bme.readHumidity(), and analog values from MQ-7 (CO) and MQ-2 (gas/smoke) sensors via analogRead() on their respective pins.

GPS Position Handling: [Details to be added if available]

Wi-Fi Connection and Data Transmission: If connected to Wi-Fi, the system prepares an HTTP POST request.

Data Format and Transmission Protocol: Sensor data (temperature, humidity, CO, gases) and GPS position are formatted into a JSON (JavaScript Object Notation) object. This JSON payload is sent via an HTTP POST request to a specific Google Apps Script URL (<https://script.google.com/macros/s/AKfycbxFWVXtDwIeyFEASvVJvReLQs2rrb7h0-VIJbcCBGjBB6fkBo-zfy95qD4vN4WM8PhFQ/exec>).

Read and Send Frequency: A 1-second delay (delay(1000)) is implemented at the end of each read-and-send cycle. Note that this differs from the earlier stated requirement of transmitting every 6 seconds; the current prototype firmware operates with a 1-second frequency.

Power Management: For this prototype, advanced energy management such as Deep Sleep is not explicitly implemented in the main loop; the delay(1000) is the only timing control, indicating continuous operation suitable for demonstrations.

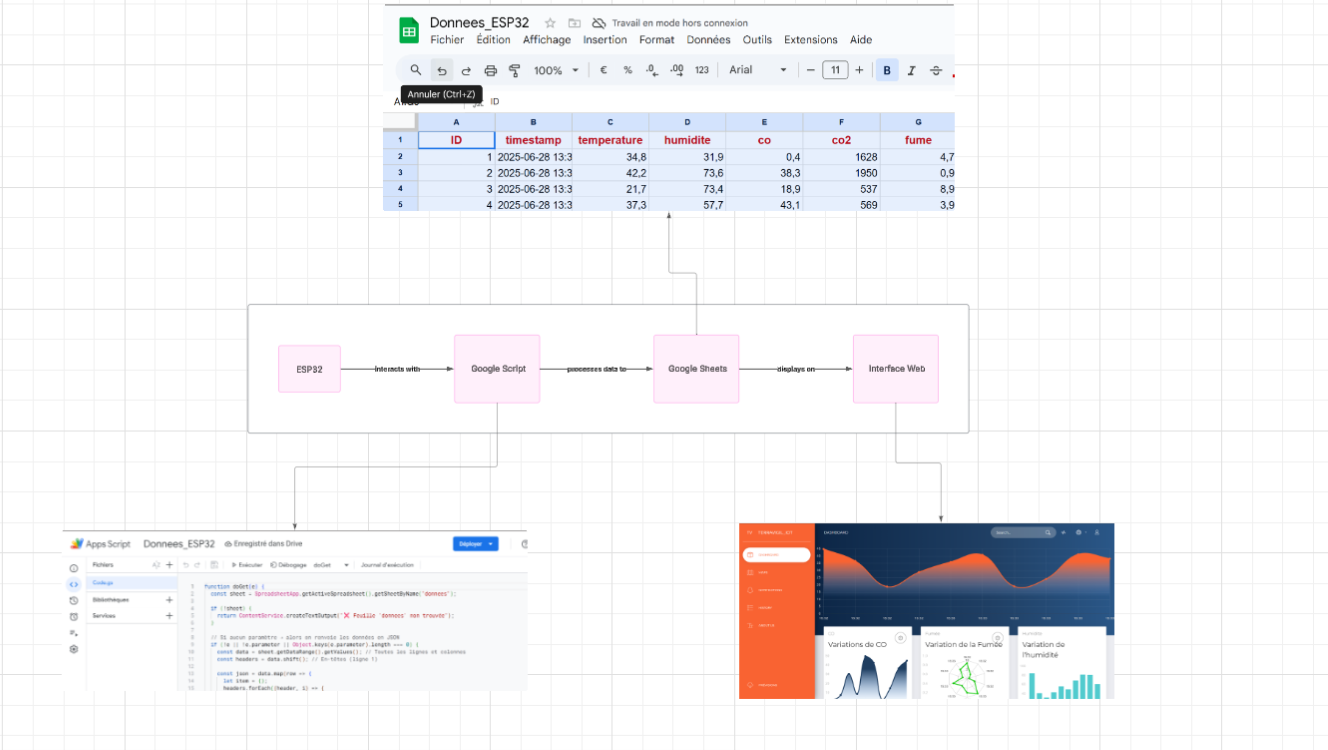
**4.2. Technologies Used**

To set up the platform, several technologies were used. These include:

|  |  |  |
| --- | --- | --- |
| Technologie | Logo | Functions |
| HTML |  | Structures the content of web pages (tables, buttons, menus, forms). |
| CSS |  | Manages visual style: colors, fonts, responsive layout and light animations. |
| JS |  | Provides page dynamics, including remote data retrieval, automatic content updating, user **Africa Deep Tech Challenge 2025** management, and integration of charts and maps. |
| Google Apps Script |  | * Processing HTTP GET requests sent by the platform. * Extracting and formatting data from the Google Sheet into JSON. * Calculating the risk level from the raw data received. * Returning data ready for use by the platform. |
| Google Sheets |  | * Google Sheets serves as a lightweight and accessible database for storing all measurements collected by IoT sensors. * Each new data point is added as a row in the spreadsheet. * Enables easy, collaborative, and real-time access to data. * Facilitates the historical storage of measurements for later analysis |
| Google Maps JavaScript API |  | This API allows you to integrate interactive maps directly into the web platform.  It is used to:   * Display the GPS coordinates of sensors as points on a geographic map. * Spatially visualize risk areas based on the detected danger level. * Offer interactive features such as zooming, navigation, and viewing information associated with each point. |

This technological architecture, combining simple, effective and complementary tools, has made it possible to quickly deploy a robust and scalable platform, perfectly adapted to the needs of early detection of bushfires.

**4.3. General architecture**



**4.5. Website architecture**

the platform is organized like this

**FireGuard**

│   ├── assets

│   │   ├── demo

│   │   ├── css

│   │   │   ├── bootstrap.min.css

│   │   │   ├── now-ui-dashboard.css

│   │   │   ├── now-ui-dashboard.min.css

│   │   │   └── now-ui-dashboard.css.map

│   │   ├── fonts

│   │   ├── img

│   │   ├── js

│   │   │   ├── now-ui-dashboard.js

│   │   │   └── now-ui-dashboard.min.js

│   │   └── scss

│   │   │   ├── now-ui-dashboard

│   │   │   └── now-ui-dashboard.scss

│   ├── docs

│   ├── examples

│   │   ├── about.html

│   │   ├── History.html

│   │   ├── notifications.html

│   │   ├── map.html

│   │   ├── dashboard.html

└── CHANGELOG.md

```

**4.6. Data Retrieval and Processing**

The Google Sheets platform, enhanced by Google Apps Script capabilities, serves as the core for processing, storing, and analyzing data received from the monitoring nodes.

1. **Backend Connection (Google Apps Script)**

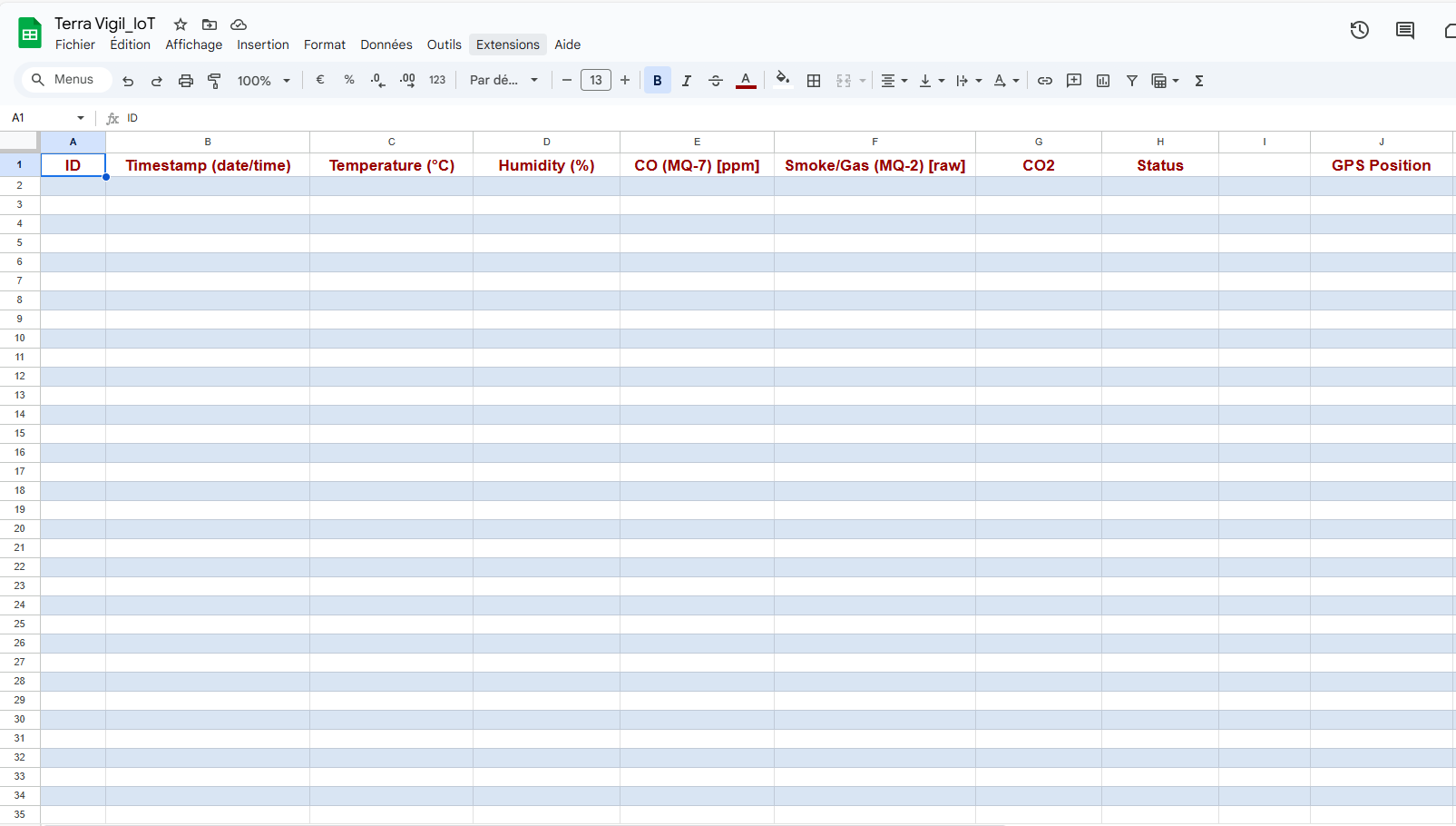
The platform retrieves data from a backend service deployed using **Google Apps Script**.  
A deployed API endpoint (using the doGet method) responds to HTTP requests and returns sensor data in **JSON** format.

* **API Endpoint**: Public URL of the deployed Apps Script.
* **Data Format**: JSON structure containing an array of measurements with fields like temperature, humidity, CO, etc.

1. **Database Structure (Google Sheets)**

The collected sensor data is stored in a Google Sheets file acting as a lightweight database.  
Each row corresponds to a new measurement, with the following columns:

* **ID**: Auto-incremented unique identifier
* **Timestamp**: Date and time the data was recorded
* **Temperature** (°C)
* **Humidity** (%)
* **CO** (ppm)
* **CO₂** (ppm)
* **Smoke**
* **GPS**: Latitude and longitude as a string
* **Status**: Calculated fire risk level (Normal, Moderate, High, Extreme)



Google Sheets Structure:

The system uses a primary Google Sheets spreadsheet. Within the doPost function, SpreadsheetApp.getActiveSpreadsheet().getActiveSheet() indicates that data is appended sequentially to the active sheet. Each new sensor reading received from the nodes corresponds to a new row in this sheet. Columns include an order number (nextNum), reception timestamp (time), sensor measurements (temperature, humidity, CO, gases), a "disabled" field, alert status (status), an empty field, and GPS position. Background colors are applied to rows to improve readability and visually distinguish entries.

Role of Google Apps Script:

The key function doPost(e) is triggered each time an HTTP POST request is received at the deployed script URL. Its roles include:

Data Reception and Parsing: It intercepts the POST content (e.postData.contents), which is a JSON object sent by the ESP node, and parses this JSON to extract temperature, humidity, CO, gas, and GPS values.

Timestamping and Indexing: It generates a timestamp (new Date()) for each entry and calculates a sequential row number (nextNum) for easy tracking.

Data Recording: The extracted data, enriched with the timestamp and index, is added as a new row in the active sheet (sheet.appendRow([...])).

Alert Detection and Status Assignment: Before saving the row, the script evaluates sensor data against predefined thresholds to determine the node’s status.

Feedback: After processing, the script returns a text response ("Success" or "Error") to the request sender (the ESP node), confirming receipt and handling.

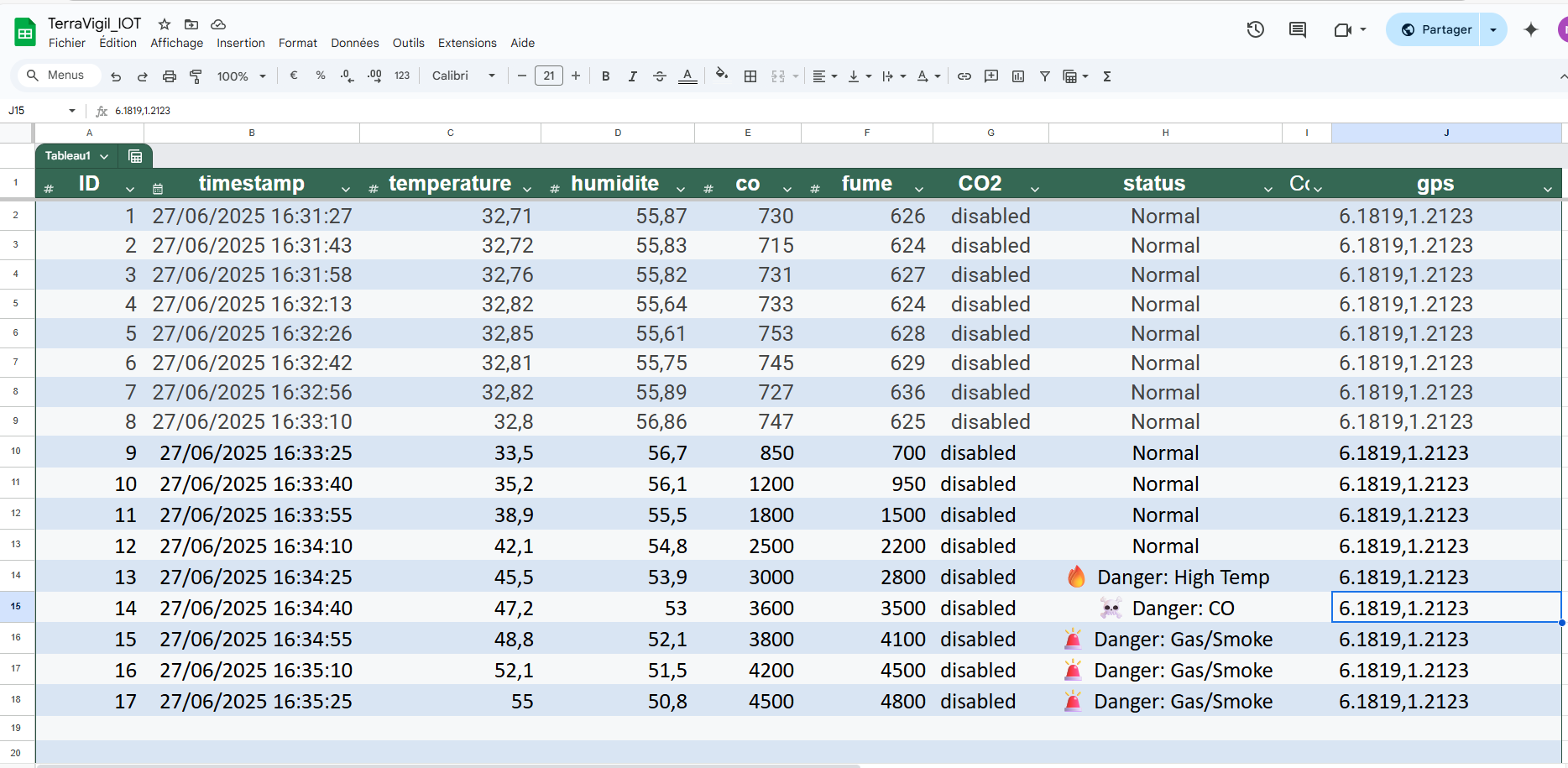
Alert Logic: Dangerous situation detection is embedded directly in the doPost(e) script via if/else if conditions. Specific alert statuses are assigned based on the following threshold breaches:

**High Temperature:** Status changes to "Danger: High Temp" if temperature > 45°C.

**CO Concentration:** Status changes to "Danger: CO" if CO value > 3500.

**Gas/Smoke Presence:** Status changes to "Danger: Gas/Smoke" if gas value > 4000.

If none of these conditions are met, the status remains "Normal". This logic enables fast, automated alerting based on sensor data.



1. **Data Fetching Method**

The data is fetched using the fetchData() function in JavaScript, which performs the following:

Sends an asynchronous request to the API endpoint. Parses and processes the returned JSON. Dynamically updates the table on the web interface. To ensure **real-time monitoring**, the data is automatically refreshed every few seconds using:

**setInterval(fetchData, 10000); Every 5 seconds**

**4.7. Data Visualization**

1. **Dynamic Table Display**

The main interface includes a dynamic and scrollable table to visualize the latest sensor data:

* Sorted by date and time (most recent first).
* Color-coded status levels using badges:
  + Green: Normal
  + Gray: Moderate
  + Orange: High
  + Red: Extreme

This design improves readability and helps users quickly identify high-risk conditions.

1. **Real-Time Data Charts**

Chart.js is used to generate interactive charts that show trends over time for each parameter:

* Temperature (°C)
* Humidity (%)
* CO (ppm)
* Smoke level

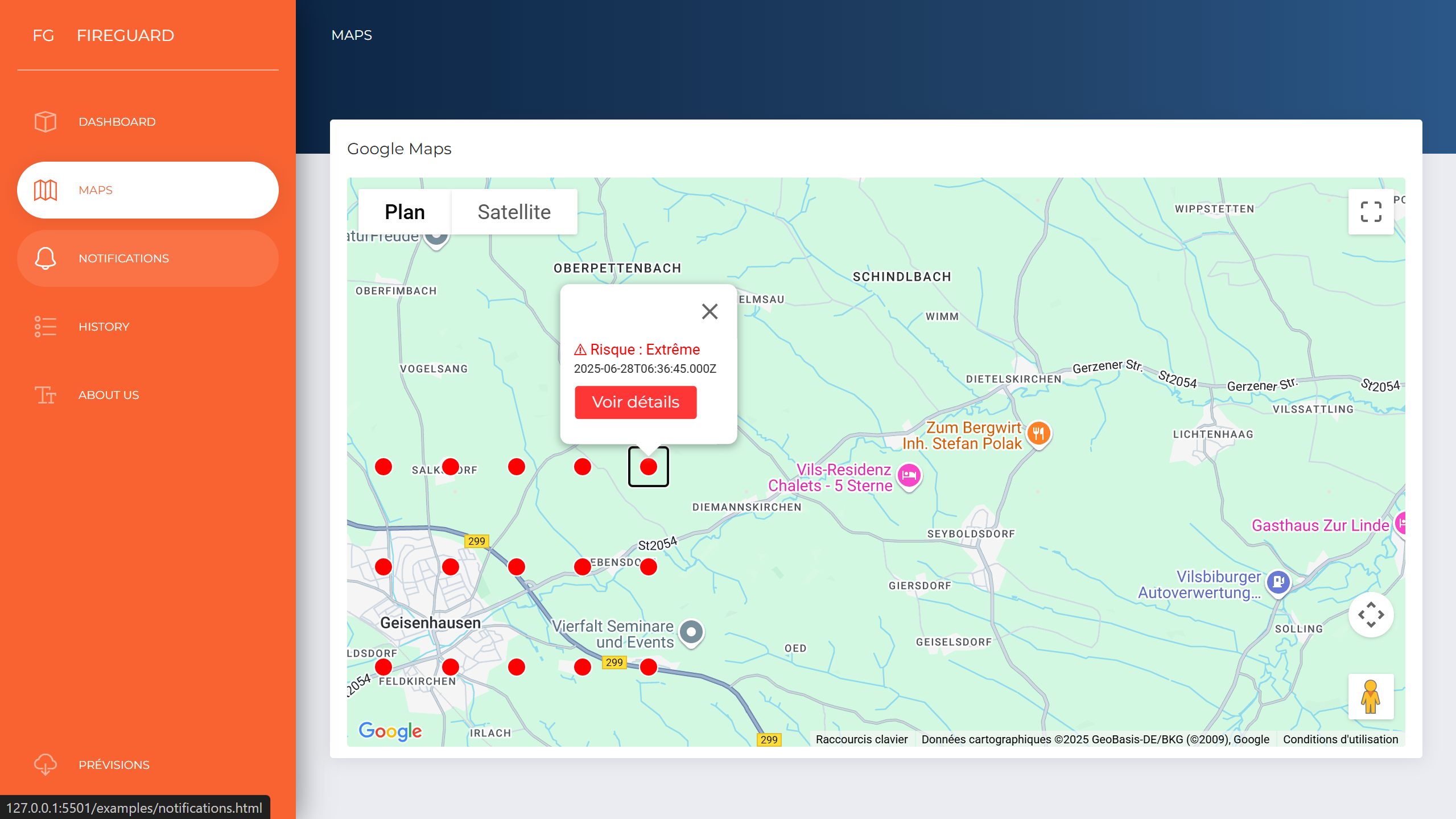
Each chart displays the last 10 values for clarity and performance. The graphs are updated every 10 seconds in sync with the data.

****

**4.8. Handling High-Risk Area**

Special attention is given to critical risk areas:

* Colored markers are used to reflect the risk level (green to red).
* The map automatically zooms and centers on the most recent critical zones.
* Tooltips display detailed information (timestamp, coordinates, risk level) when clicking a marker.

****

**5. Challenges Faced and Solutions Implemented**

The development of the **FireGuard** system was marked by several technical and conceptual challenges. Some obstacles proved to be more complex than initially anticipated, and overcoming them required careful analysis and creative problem-solving throughout the project.

**5.1. Optimizing ESP Power Consumption**

One of the main challenges was ensuring extended autonomy for the solar-powered nodes, especially during low sunlight periods or at night. This required strict control over the power consumption of the ESP32 microcontroller and connected sensors.

Although deep sleep modes were not fully implemented in the initial firmware version, several software optimizations were introduced to minimize unnecessary activity. The system was designed with a 20W solar panel and a lithium-ion battery managed by a charge controller. This configuration successfully maintained sufficient power levels for prototype operation. Further research is underway to integrate deep sleep in future iterations.

**5.2. Difficulties with Google Sheets and Apps Script Integration**

Direct communication between the ESP nodes and Google Sheets through Apps Script introduced challenges in handling HTTP POST requests, parsing incoming data, and ensuring reliable write operations. Understanding the configuration and deployment of Apps Script as a web endpoint also required time and experimentation. A reliable script was developed to handle incoming data, extract sensor readings, generate timestamps, and append entries to the spreadsheet. Error handling mechanisms were also added to catch failures and provide meaningful responses to the ESP nodes. Ensuring correct deployment permissions played a key role in stabilizing communication.

**5.3. Slow Request Performance**

Problem:  
One of the first challenges encountered was the latency in fetching and displaying the sensor data, especially as the dataset grew larger over time.

Solution:  
To address this, optimizations were applied on the client side:

* Only the 10 to 15 most recent entries are fetched and visualized in real time, instead of loading the entire dataset.
* Asynchronous JavaScript functions (async/await) are used to **Africa Deep Tech Challenge 2025** UI blocking.
* Data updates are handled incrementally with setInterval, avoiding full page reloads.

This significantly improved page responsiveness and reduced load times.

**5.4. Decimal Format Inconsistencies**

Problem:  
In some sensor readings, numeric values used a comma (",") as the decimal separator (common in French-speaking countries), while JavaScript and many APIs expect a dot (".").

Solution:  
During data processing, a conversion step was added in JavaScript to ensure proper parsing of numeric values:

parseFloat(value.replace(",", "."))

This guarantees that all temperature, humidity, CO and CO₂ values are correctly interpreted and plotted on the charts.

**5.5. Timestamp Formatting and Display**

Problem:  
Timestamps received from the backend were in raw format and not user-friendly (e.g., 2025-06-28 10:45:00). Additionally, time zones and formats caused inconsistency in visualization.

Solution:  
A formatting function was implemented to:

* Convert timestamps into localized time format using toLocaleTimeString('fr-FR').
* Ensure consistency across all components (table, chart tooltips, and map info windows).
* Add fallback mechanisms in case of corrupted or empty timestamps.

This enhanced readability and ensured consistency across the platform.

**5.6. Synchronization Between Map, Table, and Charts**

Problem:  
Displaying synchronized and coherent information across the data table, graphs, and Google Maps proved challenging — especially when new data arrived rapidly.

Solution:  
To resolve this, a unified data-fetching system was established:

* All components (table, charts, map) use the same API endpoint and data parsing logic.
* Data is updated simultaneously using the same JavaScript fetch() result.
* Proper data sorting and filtering ensure that all views reflect the same timestamp and measurements.

This ensures consistency in the user experience, and allows operators to correlate spatial (map), tabular, and temporal (graph) data at a glance.

**6. Team Contribution**

|  |  |
| --- | --- |
| **Team Members** | **Specific Contributions** |
| VLAVO Peniel Jean Bosco | Design and development of the ESP node firmware (C++ code for sensor reading and Wi-Fi communication). Hardware integration of sensors and communication modules. Unit testing of the nodes. Active participation in structuring the report. |
| ADANSOU DIANE | Design and implementation of the nodes' solar power system, including battery management and charging/regulation modules. Significant contribution to the development of the technical documentation on the hardware architecture and the system’s energy autonomy. |
| ALATE JOSUE & Joseph ESSEY | Combined contributions: Design and development of the backend architecture on the cloud platform (Google Sheets and Google Apps Script), including the implementation of data analysis logic and alert detection algorithms. Development of the user dashboard (HTML, CSS, JS frontend), including data integration for interactive visualization (maps, charts) and user interface (UI/UX) design. Joint contribution to the writing and overall revision of the project report. |

**7. Conclusion and Future Outlook**

**7.1. Project Assessment**

The **FireGuard** project successfully demonstrated the feasibility of an early-warning and environmental monitoring system for bushfires, based on low-cost, connected technology. The initial goals were met, confirming the relevance and effectiveness of the proposed approach. Key accomplishments include:

End-to-end data pipeline from sensor acquisition on the ESP node to visualization on the web dashboard, with intermediate processing and storage in Google Sheets.

Real-time alerting capability, with the system able to detect critical conditions such as high temperature, CO, and gas presence, and trigger alerts within 2 to 5 seconds.

Energy autonomy, proven through the integration of a solar panel and battery, allowing nodes to operate independently in remote areas.

Cost-effective and accessible architecture, leveraging affordable hardware components and free or low-cost cloud services.

Working prototype, providing a solid proof of concept and a strong foundation for future development.

**7.2. Future Improvements and Developments**

While the prototype is functional and promising, several enhancements are planned to scale the system and increase its resilience.

**Enhanced communication through SIM800L integration**

Implementing the SIM800L module will enable nodes or a central unit to send alert messages via SMS to emergency services, even in the absence of internet connectivity. This module can also act as a backup communication channel.

**Large-scale deployment support**

To accommodate a larger number of nodes, the architecture may be adapted to use long-range wireless technologies such as LoRaWAN or NB-IoT. Dedicated gateways and centralized management systems will also be introduced to streamline deployment and maintenance.

**Advanced data analysis using machine learning**

Predictive models can be trained on historical sensor data to identify early patterns indicating fire risk. Such models would improve responsiveness and reduce false positives. Machine learning can also help distinguish different types of combustion sources.

**Additional sensor integration**

Including a CO2 sensor would enrich the combustion analysis. Weather-related sensors, such as wind speed and direction or rainfall, would help model fire propagation and assess ground humidity levels.

**Robust outdoor hardware design**

New enclosures will be developed to withstand harsh environmental conditions, including moisture, dust, UV exposure, and mechanical shocks. Physical security features will be added to **Africa Deep Tech Challenge 2025** theft or vandalism.

**Dedicated mobile application and push alerts**

A mobile application will be developed to send real-time push notifications to users. The app will provide an optimized interface for monitoring sensor data and alerts on smartphones, including integrated map views for quick location awareness